

Python: scripting goodness

David Hostetler

Outline

- History and Philosophy
- Feature summary
- Syntax Concepts
- Datatypes/Operators
- Functions
- Namespaces
- Classes
- Exceptions
- Modules

History

- Invented by Guido van Rossum, 1990 (he had a week off around Christmas)
- Influenced by ABC, modula-3, C, C++, Icon

Python Philosophy

- **Coherence**: a scripting language ***can*** be easy to read/write/maintain.
- **Power**: embeddability and semantic power aren't mutually exclusive.
- **Scope**: useful for rapid prototyping during development and building more advanced, permanent systems.
- **Objects**: OOP actually can be useful.

```
$name =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C",  
hex($1))/eg;
```

In a nutshell

- Freeware.
- VHLL, OODL
- *NIX, DOS, OS/2, WINxx, Mac, etc..
- Used for:
 - Shell scripting
 - GUIs
 - Database access
 - Pure prototyping
 - Internet scripting (CGI, HTTP, etc..)
 - Distributed programming
 - **Extensions: frontends, customizations to C/C++ libraries**

Features in a nutshell

- No compile or link steps
- No type declarations
- Automatic memory management
- OOP
- Classes, modules, exceptions
- Dynamic module loading/reloading
- Universal, 'first-class' object model

Core Syntax Concepts

- End of block syntax: The indentation IS the block indicator. No Brackets!

```
if notSafe():
    if condition():
        dosomething()
        dosomething2()
    elif x > y:
        doadifferentthing()
    else:
        x = y
else:
    quit()
```

Core Syntax Concepts

- End of statement syntax: generally, the end of the line IS the end.

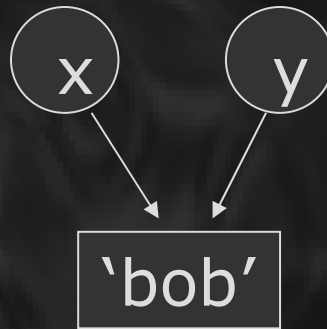
```
if notSafe():           # this is a comment
    dosomething()
    dosomething2()
elif (a_really_long_condition and
      another_condition):
    weekend = 'Saturday' + \
             'Sunday'
else:
    x=y; a='bob'; z=3.14159
```

Core Syntax Concepts

- Everything is a reference to a (potentially shared) object.

```
x = 'bob'
```

```
y = x
```



Since assignment can create sharing, you can use `'is'` to test if two variables are sharing the same object, as opposed to just having equivalence:

```
if x is y:
```

```
    y = x[:]
```

```
# now x and y are equivalent, but not the same object
```

5 June 2001

Data Types

- Numbers
- Strings
- Lists
- Dictionaries
- Tuples

Data Types: Numbers

Value:

- 123
- 3.14159
- 0123
- 0x123
- 999999999999L
- .2e-5

Interpretation:

long int

double

octal

hex

long (infinite)

double

Data Types: Strings

Strings are immutable. There is no char.

String:

- ''
- 'x'
- "bob's string"
- 'a "foo" string'
- 'ho'*3+'humbug'
- 'bob\'s %d %s' % (2,'dogs')
- ""Once upon a time...
...the end.""

Interpretation:

empty string

x

bob's string

a "foo" string

hohohohumbug

bob's 2 dogs

A block of text

Data Types: Lists

Lists are arrays of object references.

- Grow/shrink on demand
- Are mutable
- Support nesting
- Can contain any kind of object

List:

- []
- [x]
- [3,['a','bc']]
- list[i]=1.4
- list+2*list
- list.append(y)
- del list[z]

Interpretation:

empty list
single item
nested, mixed
index, assignment
concatenation, repetition
growing via method
shrinking

Data Types: Dictionaries

Dictionaries are associative arrays.

- Mutable mappings
- Entries are stored/fetched by key
- Almost any datatype can be the key
- Just dynamically sized hash tables
- Can contain any kind of object

Expression:

- `{}`
- `{'dogs':3, 'cats':2}`
- `{'green':{'eggs':'ham', 'bob':3.2}}`
- `foo['dogs'] = 4`
`foo['green']['eggs']='samIam'`
- `foo.has_key('bob')`
- `stuff = foo.keys()`

Interpretation:

- empty dictionary
- two entries
- nested
- index, assignment
- test for existence
- get a list of keys

Data Types: Tuples

Tuples are sequences of object references.

- Immutable!
- Analogous to lists, except for the immutable part

Tuple:

- ()
- (x,)
- (x)
- (a,b,(1,2,'spoon!'))
- (x,y,z)=(2,4,8.2e-10)
- (x,y,z)=foo
- foo,bar=bar,foo
- tuple[i]=x
- x = tuple[i]
- joe+bob, bob*8

Interpretation:

- empty tuple
- single item tuple
- Expression, NOT a tuple!
- nested
- assignment
- foo must be a 3 item tuple
- a swap with no temp!
- Error!!
- ok.
- concatenation, repetition

Operators

Just like C, except:

Use:

- and
- or
- not

Instead of:

&&
||
!

Also:

- is, is not
 - in, not in
 - $x < y \leq z$
 - if (x=y)
 - No support for --, ++, +=, ?:, etc..
- like pointer comparison
sequence membership
support for chaining
Error! '=' is not an operator.

Slicing

Range of indices of an array:

```
str = 'abcdefg'
```

slice:

- `str[:]`
- `str[1:]`
- `str[:1]`
- `str[:-1]`
- `str[2:5]`
- `str[-4:-2]`

yields:

```
'abcdefg'  
'bcdefg' _  
'a' _  
'abcdef'  
'cde'  
'de'
```

Works for strings, lists, tuples.

-- Pause --

- ~~History and Philosophy~~
- ~~Feature summary~~
- ~~Syntax Concepts~~
- ~~Datatypes/Operators~~
- Functions
- Namespaces
- Classes
- Exceptions
- Modules

Functions

```
def somefunction(x, y):  
    z = x + y  
    return z
```

```
val = somefunction('joe', 'bob')
```

- Arguments are 'passed by object reference':
Meaning there's an implicit assignment:
`val=somefunction('joe','bob') → x='joe' y='bob'`

Functions

Variable Number of Arguments:

```
def dostuff(x,y,*atuple):  
    size = len(atuple)  
    return size, x%y, size*atuple(2)
```

```
a,b,c=dostuff(1,2,3,4,5,6)
```

Functions

Default Argument Values:

```
def dostuff(x,y=4,filename='cfg.txt'):
```

```
    ...
```

```
dostuff(5.23)
```

```
dostuff(5.23, filename='a.txt', y=8)
```

Namespaces

Only 3 scopes, ever:

In search order: **local, global, built-in**

```
x='bob'  
def dostuff(y):  
    y=x*4           # x is found in global  
    x=y*4           # Error! Already used x as global  
                   # must declare global to assign  
  
def dostuff(y):  
    global x        # now we can assign to it  
    x=y*4
```

Namespaces DO NOT NEST – i.e. a function's parent scope is not directly accessible.

Classes

Members and methods are virtual and public.

```
class foo(bar, bar2):  
    somemember=3  
    def somefunction(self, x, y):  
        self.somemember = x + y  
        return self.somemember
```

What's this
'self' thing?

```
x=foo()  
x.somemember=3.26  
x.somefunction('joe', 'bob')
```

foo.somefunction(x, 'joe', 'bob')

Classes

Special Methods:

```
class complex():
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
    def __del__():
        # destructor
```

```
x=complex(3,-4.2)
>>>print x.r,x.i
(3,-4.2)
```

A note about namespace scope for classes: local names are resolved from the bottom up, starting at the instance and moving through the class inheritance hierarchy.

Exceptions

```
import string, sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(string.strip(s))
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

Exceptions

Defining your own exceptions:

```
class MyError:
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return `self.value`

try:
    raise MyError(2*2)
except MyError, e:
    print 'My exception occurred, value:', e.value
```

My exception occurred, value: 4

5 June 2001

26

Modules!

```
Class argv:  
    ...
```

File: `sys.py`

Statement:

```
import sys  
from sys import argv  
from sys import *
```

usage:

```
sys.argv  
argv  
argv *
```

If you imported the module as a named object (import sys) then you can force a reload:

```
reload(sys)
```

Stuff I Left Out:

- Keyword arguments
- Inherent support for imaginary numbers
- Static vars for class objects
- Common modules
- File I/O stuff
- ??